

ACM Pacific NW Region Programming Contest
11 November 2000

PROBLEM A
The Quadratic formula (Mod p)

The Quadratic Equation was a topic that preoccupied you for some time in Algebra. In this problem you'll be revisiting this topic. You are to find the roots of a quadratic equation:

$$a x^2 + b x + c \equiv 0 \pmod{p}$$

with a little twist. The coefficients a , b and c as well as x are positive integers in the range $1 \dots p$ where p is an odd prime number (that is included as part of the input). All operations are done (Mod p). For instance, consider the equation $3 x^2 + 1000 x + 65709 \equiv 0 \pmod{p}$ and $p = 337639$. (You may trust us, 337639 is a prime number). You may verify $x = 2345$ is one root of this quadratic (Mod p).

One way to solve a modular quadratic is to use the good old Quadratic formula. The only caveat is how to perform the operations needed in the quadratic formula (**efficiently!**). For example, we need:

- the "power" (Mod p) operation,
- the "square root" (Mod p) operation, and finally
- the "division" (Mod p) operation.

Modular Power Operation

Modular power is defined by the equation: $(a)^b \pmod{p}$ You take the exponent of the number a and calculate the result (Mod p):

Examples of Modular Power Operation:

- CASE: $p = 7$:
 $(5)^4 \equiv 625 \equiv 2 \pmod{p}$
- CASE: $p = 13$:
 $(5)^4 \equiv 625 \equiv 1 \pmod{p}$

Modular Square Root Operation

A number n has two square roots (Mod p), if and only if the following condition holds:

CONDITION	IMPLIES THAT
$n^{((p-1)/2)} \equiv 1 \pmod{p}$	n has two square roots (Mod p)
$n^{((p-1)/2)} \not\equiv 1 \pmod{p}$	n has no square root Mod p

If n has two "square roots" (Mod p), then there exists two integers r_1, r_2 such that:
 $n \equiv r_1^2 \pmod{p}$ and $n \equiv r_2^2 \pmod{p}$ The main trick is finding the integers r_1, r_2 . While we **won't** show you how to calculate a "square root" (Mod p), (that's your job!) we will show you how these square roots work:

Examples of Modular Square Root Operation:

- CASE: $p = 7$:
Given below are the two "square roots" of 4 (Mod p)
 $r_1 = \text{Sqrt}(4) \equiv 2 \pmod{p}$
 $r_2 = \text{Sqrt}(4) \equiv 5 \pmod{p}$
Make sure these are indeed the "square roots" of 4 (Mod p)
Check r_1 : $(r_1)^2 \equiv 2 * 2 \pmod{p} \equiv 4 \pmod{p} \equiv 4 \pmod{p}$
Check r_2 : $(r_2)^2 \equiv 5 * 5 \pmod{p} \equiv 25 \pmod{p} \equiv 4 \pmod{p}$
- CASE: $p = 337639277$:
Given below are the two "square roots" of 17 (Mod p)
 $r_1 \equiv \text{Sqrt}(17) \equiv 113622037 \pmod{p}$
 $r_2 \equiv \text{Sqrt}(17) \equiv 224017240 \pmod{p}$
Make sure these are indeed the "square roots" of 17 (Mod p)
Check r_1 : $(r_1)^2 \equiv 12909967292029369 \equiv 17 \pmod{p}$

ACM Pacific NW Region Programming Contest

11 November 2000

Check r_2 : $(r_2)^2 \equiv 50183723817217600 \equiv 17 \pmod{p}$

Modular Division Operation

In order to do modular division, you need to understand the modular multiplicative inverse operation. Assume that z is the multiplicative inverse of a number b then the following should hold:

$$z \cdot b \equiv 1 \pmod{p}$$

this implies that

$$z \equiv (b)^{-1} \pmod{p}$$

thus, $(b)^{-1}$ is the multiplicative inverse of b . To divide any number a by $b \pmod{p}$ simply multiply a by the multiplicative inverse of b

Examples of Modular Division:

CASE: $p = 7$:

Calculate $5/4 \equiv 5 \cdot (4)^{-1} \pmod{p}$

First, find the inverse of $4 \pmod{p}$: $(4)^{-1} \equiv 2 \pmod{p}$

Second, calculate $5 \cdot (4)^{-1} \equiv 5 \cdot 2 \equiv 10 \equiv 3 \pmod{p}$

Check: $(5/4) \cdot 4 \equiv 3 \cdot 4 \equiv 12 \equiv 5 \pmod{p}$

CASE: $p = 13$:

Calculate $5/4 \equiv 5 \cdot (4)^{-1} \pmod{p}$

First, find the inverse of $4 \pmod{p}$: $(4)^{-1} \equiv 10 \pmod{p}$

Second, calculate $5 \cdot (4)^{-1} \equiv 5 \cdot 10 \equiv 50 \equiv 11 \pmod{p}$

Check: $(5/4) \cdot 4 \equiv 11 \cdot 4 \equiv 44 \equiv 5 \pmod{p}$

The Program

!!!NOTE!!!: Calculations **may require integers** up to a maximum of **64 bits** in length.

Your task is to write a program that reads quadratic equations from a text file (**a.dat**), and determines whether or not each of the equations in the input has roots **(Mod p)**. Each quadratic equation is on a separate line. The coefficients **a, b, c** of each quadratic equation and a modulus **p** are given on each line. You may safely assume that all the non-negative values of **p** are odd prime numbers, however, if you encounter a negative **p** value, you should output the message **"invalid input"** as shown below in the sample. Your **program must be efficient**, because the **input file will contain a large number of equations** to solve. For each equation, output the equation and the root(s) in the following format:

```
Q[x_] := Mod[ax^2 + bx + c, p ]
{ root(s) or message goes here }
```

...blank line...

```
Q[x_] := Mod[ax^2 + bx + c, p ]
{ root(s) or message goes here }
```

...blank line...

To see how this format corresponds to actual input look at the sample input and output given below.

<u>Sample Input</u>	<u>Sample Output</u>
4 3 3 -13	Q[x_] := Mod[4x^2 + 3x + 3, -13]
4 3 3 13	{ invalid input }
17 8 1 71	
3 1000 65709 337639	Q[x_] := Mod[4x^2 + 3x + 3, 13]
1 179344794 146367396 179424691	{ 11 }
	Q[x_] := Mod[17x^2 + 8x + 1, 71]
	{ has no roots }
	Q[x_] := Mod[3x^2 + 1000x + 65709, 337639]
	{ 2345, 109868 }
	Q[x_] := Mod[1x^2 + 179344794x + 146367396, 179424691]

ACM Pacific NW Region Programming Contest
11 November 2000

{ 78021, 1876 }

ACM Pacific NW Region Programming Contest
11 November 2000

PROBLEM B
Base Addition

Given 3 values, s_1 , s_2 , s_3 , what is the smallest integer having a base less than or equal to 36 such that $s_1 + s_2 = s_3$? Each string, s_1, s_2, s_3 , is composed of digits and upper case letters, with ASCII value ordering (0123...XYZ).

INPUT:

Groups of 3 strings (one per line) s_1 , s_2 , s_3 .
Each string will be at most 80 characters.
Each string will consist entirely of digits & upper-case letters.
Input will end with a single 0 alone on a line.
Thus, the total number of lines will be $3*k + 1$ (for some integer k).
There will be no leading zeros.
Input file for this problem is **b.dat**

OUTPUT:

For each triplet of s_1 , s_2 and s_3 , print one line of output (line-wrap is allowable for long strings):
If a value, x , is found that meets the criteria, print: $s_1 + s_2 = s_3$ in base x
If no answer exists, print: $s_1 + s_2 \neq s_3$

SAMPLE INPUT:

```
5
3
10
5
3
11
5
3
8
5
3
7
ABC
DEF
18AB
11111111111111111111
22222222222222222222
33333333333333333333
0
```

SAMPLE OUTPUT:

```
5 + 3 = 10 in base 8
5 + 3 = 11 in base 7
5 + 3 = 8 in base 9
5 + 3 != 7
ABC + DEF = 18AB in base 16
11111111111111111111 + 22222222222222222222 = 33333333333333333333 in base 4
```

**ACM Pacific NW Region Programming Contest
11 November 2000**

**ACM Pacific NW Region Programming Contest
11 November 2000**

PROBLEM C Solitaire Cribbage

Cribbage is an old card game, usually played by 2 people. Points are scored by having various card combinations in a hand. In Cribbage, many of these combinations are similar to other card games, two-of-a-kind, three-of-a-kind, four-of-a-kind and runs (numerically sequential cards of 3 or more). But also in Cribbage, combinations of cards that add up to 15 are important. (Face cards are counted as 10 points, and aces are worth 1 point).

In a standard, two-player game, each person is dealt 6 cards. Four of these cards are kept as the player's hand, and the other 2 are placed in the "crib". The crib is given to each player alternately. Therefore, scoring is done on groups of 4 cards (either in the hand or the crib). In addition, a single card is turned up (after the crib is made), which can be used to supplement the scoring.

In our version of Cribbage, we will be "dealt" 8 cards, plus the one extra. It is the job of your program to divide the 8 dealt cards into a hand and a crib (4 cards each) such that the maximum total score is achieved, using the extra card as appropriate. (That is, the sum of the hand and the crib is maximized.)

For our purposes, we will not use card suits, and we will represent the cards as single characters, as follows:

Ace=1, Two=2, Three=3, Four=4, Five=5, Six=6, Seven=7, Eight=8, Nine=9, Ten=T, Jack=J, Queen=Q, King=K

Scoring

Two-of-a-kind (a pair): 2 points

Three-of-a-kind (3 pair): 6 points

Four-of-a-kind (6 pair): 12 points

Runs (sequentially numbered cards of length greater than 2): 1 point per card in the run

Fifteens: 2 points for any distinct combination of cards that sum to 15

Example #1

Hand dealt: 5T524765

Extra card: 9

Best possible total score: 20 points

Explanation:

Hand: 555T = 14 points (3 pair for 6 points, plus 4 15s for 8 points(10+5, 10+5, 10+5, 5+5+5))

Crib: 2467 = 6 points (3 15s (9+6, 2+4+9, 2+6+7))

(Note that either group of 4 could be the "crib" and either group the "hand".)

Example #2

Hand dealt: 46K98827

Extra card: 5

Best possible total score: 18 points

Explanation:

Hand: 6788 = 14 points (2 runs of 4 cards for 8 points, a pair of 8s for 2 points, and two 15s (8+7,8+7) for 4 points)

Crib: 249K = 4 points (2 15s (2+4+9, 5+K))

ACM Pacific NW Region Programming Contest

11 November 2000

Example #3

Hand dealt: 76547T61

Extra card: 6

Best possible total score: 25 points

Explanation:

Hand: 4566 = 21 points (3 runs of 3 cards for 9 cards, 3 pair for 6 points, 3 15s for 6 points)

Crib: 177T = 4 points (a pair for 2 points, one 15 (1+7+7) for 2 points)

Note that the extra card is not displayed as a part of either the hand or the crib, but it can be used in scoring for both.

Your task is to write a program that reads a file of “dealt” cards, and returns the maximum point total for each dealing, along with the hand and crib configuration that produced that score. (Order of cards in display is ascending. Which group of 4 is the hand and which is the crib is immaterial.)

Input:

The input file will consist of a series of lines, each line containing 9 “cards”. The last card on each line will be considered the “extra” card for that dealing. The last line in the file will consist of 9 zeroes, which will indicate the end of file, and the end of your processing. All input characters will be of a valid card type, as explained above. Cards represented by alphabetic characters will be capitalized. There will be no spaces between card characters, and no blank lines in the file.

The input file for this problem is **c.dat**.

Output:

```
Deal #1:
Extra Card: x
Hand: xxxx = yy points
Crib: xxxx = yy points
Best Score = zz
```

```
Deal #2:
Extra Card: x
Hand: xxxx = yy points
Crib: xxxx = yy points
Best Score = zz
```

etc.

Note that a line of whitespace is *required* between the output for each dealing.

Note also that the explanation of scoring (as seen in the examples) is not to be displayed in the actual program output.

ACM Pacific NW Region Programming Contest
11 November 2000

PROBLEM D
Poker Solitaire

This problem features a variant of the game of "poker solitaire" (PS). To play PS, deal a 5x5 grid of cards from a standard deck. The score for a PS grid is the SUM of the scores for each row, column, and diagonal. These twelve hands each score as follows:

- pair (1 point) two of the five cards have the same rank.
- two pair (3 points) two pairs amongst the five cards.
- three of a kind (5 points) three of the five cards have the same rank.
- straight (7 points) the five cards may be arranged in sequence, 8--6--9--7--5. Aces may be high (above King) or low (below 2).
- flush (10 points) All five cards are the same suit.
- full house (12 points) one pair and three of a kind.
- four of a kind (25 points) four of the five cards have the same rank.
- straight-flush (50 points) both a straight and a flush.

In normal poker solitaire, the player deals one card at a time and can place the card in any open space until all twenty-five are dealt. But we've changed the game! We'll give you a grid of twenty-five cards and then you'll swap pairs of cards to try to maximize the score. At each step, choose the pair of cards that maximizes the score (be greedy).

You'll need to calculate the number of steps until no single step yields improvement.

If there are several possible steps with the same score, break ties by choosing the pair with the lowest POSITIONED card in it, using the following grid:

```
1  2  3  4  5
6  7  8  9 10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25
```

For example, swapping 1 & 24 would take priority over swapping 2 & 3. If there's still a tie (for example, 1 & 3 vs. 1 & 5), use the second card's position in the grid, taking the lower one.

INPUT:

The input file will contain a series of 5x5 grids of cards. Each line will correspond to a row of five cards. Each card will be a rank ('2'-'9', 'A', 'K', 'Q', 'J', 'T') followed by a suit ('C', 'D', 'H', 'S'). For example, a single row would look like this:

```
TD 2S 3C AS QH
```

Each row will be flush to the left and cards will be separated by exactly one blank. Thus there are EXACTLY 14 characters per row.

There will be a row with 14 zeros ("00000000000000") to signal the end of input.

The number of rows will be divisible by 5 (we won't give you a partial grid at the end).

**ACM Pacific NW Region Programming Contest
11 November 2000**

SAMPLE INPUT:

```
AD KD QD JD TD
AC TC QC JC KC
AS KS QS JS TS
AH KH QH JH TH
9S 9C 9D 9H 8S
AD KD QD JD TD
TC AC QC JC KC
AS KS QS JS TS
AH KH QH JH TH
9S 9C 9D 9H 8S
AC KD TH JD TD
TC AD QC JC KC
AS KS JS JH TS
AH KH QH QH 9D
9S 9C QD 9H 8S
000000000000000
```

OUTPUT:

For each grid, print three lines, followed by a blank line, with:
the score for the initial grid
the number of steps
the score for the final grid

OUTPUT for SAMPLe INPUT:

```
Initial score = 317
Steps = 1
Final score = 357
```

```
Initial score = 298
Steps = 2
Final score = 357
```

```
Initial score = 38
Steps = 7
Final score = 314
```

**ACM Pacific NW Region Programming Contest
11 November 2000**

PROBLEM E Embedded Codes

Before the age of computing, some of the simplest codes were sent in plain view, embedded in a long string of text. The simplest type of this embedded code is to “hide” a string of text every ‘n’ characters in the larger block of text. The recipient only needed to know the value of ‘n’, to extract the message.

You are to write a program that searches a block of text for a given string. Determine if the string is embedded *somewhere*, and if so, report the ‘n’ value.

For example,

String to search for: Hello World

Text to search through: AHaealalaoa aWaoaralad

Result: “Hello World” is found with encoding of 2.

In this problem, case matters. Treat all characters in the string to search for as significant, including spaces. That is, in the above example, if the text to search through was AhaealalaoaWaoaralad, then “Hello World” is not found.

INPUT: The input file for this program will consist of a series of search pairs. The first line of such a pair will be the string to search for (the embedded code). This line will be no more than 80 characters long and will be terminated by the character “*”. The next (up to) 255 characters will be the text to search through. The character “*” will determine the end of this line. Both the search string and the text to search through will be comprised of alphanumeric characters and the space character. There will be no punctuation, carriage return/line feeds or any other whitespace other than the space character contained in either string. (The file, of course, will contain carriage return/line feeds, but these will not be found in either string.) The end of the input file will be denoted by the “#” character. You may assume the text to search through is at least as long as the string being searched for.

The input file will be **e.dat**.

OUTPUT: For each search pair, output one line of text, either:

[search string] is not found.

Or

[search string] is found with encoding of n.

where “search string” is replaced with the actual string being searched for, and “n” is replaced with the integer encoding value.

EXAMPLE I/O:

INPUT

OUTPUT

Hello World*	[Hello World] is found with encoding of 2.
Ahbecldleof gWhoirjlkd*	[Hello World] is not found.
Hello World*	[DOS RULZ] is found with encoding of 3.
AhbecldleofgWhoirjlkd*	[DOS RULZ] is not found.
DOS RULZ*	
ZaDerOsss87 poRkjUaaL9lZ*	
DOS RULZ*	
ZaDerOsss87 poRkjUaaL9lZ*	
#	

ACM Pacific NW Region Programming Contest
11 November 2000

PROBLEM F
Radio Transmitters

Congratulations! You've just taken a job as an analyst for KACM radio, a station that broadcasts to a region extending over the square from [-10,-10] to [10,10] inclusive.

KACM has some (one or more) transmitters of varying power, all located at integer coordinates inside the broadcast region. They would like to have their signal strength exceed a constant at every integer coordinate in the zone. The signal strength at a point is just the sum of the signals from all transmitters. This total signal must exceed 100.

The signal at location L from a transmitter T of power P is given by:

$$\text{floor}(P / D(L,T)^2)$$

That is, signal degrades quadratically in this world.

Sometimes the signal at all points is above the threshold, but usually it isn't. In such cases, KACM wants to know where to build a single new tower with sufficient (but not excess) power to cover the region. Your job is to determine the integer X,Y location and integer power P such that the signal at all locations within the region exceeds 100.

Note that it is possible to place more than one transmitter at a single location. If such is the case, the power for each transmitter is evaluated separately.

INPUT:

A series of transmitter lists. Each list consists of X Y P integer triples. The end of each list is a "0 0 0" triple, and the series ends with an empty group (i.e., an extra "0 0 0").

The input file is **f.dat**.

OUTPUT:

For each transmitter list, your program is to determine the location of the lowest-powered transmitter that can be added to the grid to provide the needed signal strength at all locations on the grid. If more than one location can be used for such a transmitter, select the location with the smallest X coordinate. If more than one location exists with the same X coordinate, select the location with the smallest Y coordinate. Print a line indicating the transmitter power required and selected location using a format similar to that of the sample output. If every location on the grid already receives a signal of adequate strength, print the message "No additional transmitters needed" on a separate line.

ACM Pacific NW Region Programming Contest

11 November 2000

Sample INPUT:

```
0 0 20000
0 0 0
0 0 10000
0 0 0
5 5 5000
-5 5 5000
0 0 0
7 7 1224
7 -7 1224
-7 7 1224
0 0 0
7 7 2000
7 -7 2000
-7 7 2000
0 0 0
5 2 1000
7 6 3000
-7 5 2000
-10 10 300
10 -10 5000
-10 -10 3000
0 0 0
0 0 0
```

Sample OUTPUT:

```
No additional transmitters needed
Add a power 10000 transmitter at 0,0
Add a power 6900 transmitter at 0,-10
Add a power 9990 transmitter at -3,-3
Add a power 6120 transmitter at -4,-4
Add a power 169 transmitter at -1,3
```

Note:

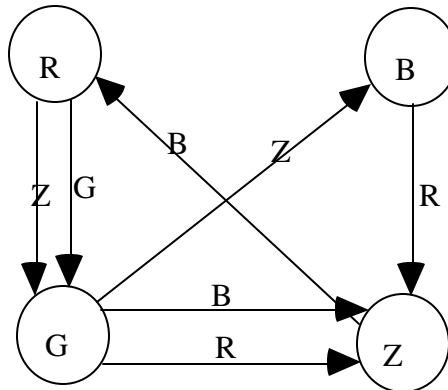
The sum S at any location is guaranteed to be less than 1,000,000 (we're not going to throw a bunch of high-powered transmitters at you and watch your program overflow).

ACM Pacific NW Region Programming Contest
11 November 2000

PROBLEM G

Color Circles

A very colorful one-person game can be played as follows. First a set of colors is selected then a set of circles is drawn using some or all of the colors, with duplicates possible - there are at least as many circles as colors. These circles are then connected together in any way by colored arrows- any number of arrows, with any colors, may be used to connect any pair of circles. For example, if we use the four colors R, G, B, and Z and four circles then we could have the following situation:



Three different circles are then picked from the set; two of them have a counter placed inside, while the third is the "target" circle. A counter may be moved from one circle to another along an arrow (in the correct direction), only if the other counter is not in the circle being moved to, and the color of the arrow is the same as the color of the circle the other counter is in. A counter may be moved several times in succession - they don't have to be moved alternately. The aim is to get one of the counters in the target circle, in the least number of moves; if the target circle can't be reached, the game is "impossible".

For example, in the picture above, if one counter is in the B circle, the other counter is in the Z circle, and the target is the G circle, then the game can only be won by moving the Z counter to the R circle (since a B arrow runs in that direction), which makes it possible to move the B counter to the Z circle along the R arrow, and the R counter can now be moved to the G circle along the Z arrow, for a total of three moves.

Input will be from a text file and will consist of descriptions of several games, using numbers instead of colors. The first line of each game description contains five numbers, N, R, S, T, M where N is the number of circles in the game (they will be numbered 1 to N, with $N \leq 100$), R and S are the numbers of the circles the two counters start in, T is the number of the target circle, and M is the total number of arrows connecting the circles ($M \leq 5,000$). After this are several lines (maximum length 60 characters) containing N numbers giving the colors of the circles in order (1 to N), with up to 20 numbers per line, separated by one or more spaces. The colors are denoted by numbers from 1 to N - some of these numbers may be unused. Then come M lines which define the arrows, in no particular order. Each contains three numbers; the first is the number of the starting circle, the second the number of the ending circle, and the third is the color of the line. The input will be terminated by a line consisting of a five zeroes. The first example below describes the picture above.

ACM Pacific NW Region Programming Contest

11 November 2000

Output, which must be written to standard output (the screen), must be one number for each game description giving the minimum number of moves to complete the game, or 0 if the game is impossible.

The data file is called **g.dat**.

EXAMPLE

Input

```
4 2 3 4 7
1 2 3 4
1 4 3
1 4 4
2 3 1
3 1 2
4 2 3
4 3 2
4 3 1
5 3 4 1 8
2 3 2 1 4
2 1 2
4 1 5
4 5 3
5 1 4
3 2 1
3 2 2
5 3 3
3 5 1
0 0 0 0 0
```

Output

```
3
4
```

**ACM Pacific NW Region Programming Contest
11 November 2000**

PROBLEM H SwampNet Routing

Your competitors in the low end routing business are adding firewall type features and your sales department is frantic. They want to add access lists ASAP.

But the marketeers also require that 'wire speed' routing not be compromised and the routing engine is underpowered. Your job is to write a simulator to help answer the performance questions.

Engineering has used a sniffer to record network traffic and extracted the relevant information from the packet headers.

Here are some samples of the simulated traffic packets:

```

216.35.137.204      0  131.215.211.5      0  1
131.215.139.100 49152 131.215.90.109    53 17
 24.218.179.217 27015   131.215.89.4    2326 17
148.246.129.175  6699 131.215.142.85   1138 6
  
```

The fields are:

source IP address	source port	destination IP address	destination port	protocol
dotted decimal address	decimal number	dotted decimal address	decimal number	decimal number

The fields are separated by one or more blanks and the first field may be preceded by one or more blanks.

A dotted decimal address is a 32 bit value, high order byte first, with the decimal representation of each 8-bit byte (0..255) separated by a '.' (decimal point). For example, 1.2.255.15 is 0x0102FF0F.

Ports are in the range: 0..65535

Note that ports are only significant for the TCP (6) and UDP (17) protocols. The port fields will be 0 for other protocols.

The protocol is in the range: 0..255

You will be given sample access lists. They will be syntax checked with fields that are separated by one or more blanks. Some example entries:

```

* 1.2.3.4 255.255.0.15 3 5.6.7.8 0.0.255.255 9 Permit
6 0.0.0.0 0.0.0.0 * 131.215.254.254 255.255.255.255 21 deny
  
```

The fields are:

protocol	source IP address	source mask	source port	destination IP address	destination mask	destination port	action
decimal number or '*'	dotted decimal address	dotted decimal mask	decimal number or '*' or a range	dotted decimal address	dotted decimal mask	decimal number or '*' or a range	'permit' or 'deny' (case not significant)

ACM Pacific NW Region Programming Contest

11 November 2000

A range is two decimal numbers separated by '-' with no whitespace between the numbers and the '-'. For example, 5-52 will match any port number between 5 and 52.

In the access list, the protocol and the ports can be specified as '*', which means match any protocol or port.

A dotted decimal mask is the same format as an address. It indicates which bits are significant when comparing addresses. For example, the addresses 123.234.248.14 and 123.234.254.126 match when compared with the mask 255.255.240.15 (0xFFFF00F). The 1 bits in the mask show which bits in the addresses to compare. Note that a mask of 0.0.0.0 will cause any address pair to match.

The way an access list works is each simulated traffic packet is compared with the access list entries, in the same order those entries were provided, until the packet matches an entry. For a match to occur, the protocol, source and destination addresses (using the respective masks), and ports must all match. In a real router, the action directs the router to accept or reject the packet, but the simulator just keeps track of how many access list entries are compared, and the number of times each action is taken.

For example, if a packet matches the 3rd, 5th, and 8th entries in a 30 entry access list, (with the action of the 3rd entry being 'permit'), then processing that packet will contribute 3 comparisons and add 1 to the number of 'permit' actions taken. In this case the 5th and 8th entries are not examined further. The first entry that matches is the last examined for a given packet.

INPUT:

The input file will be a series of test cases. Each test case consists of two groups of data, separated by a blank line. A blank line also separates test cases. The first group of data in a test case is the access list entries, one per line. There will be no more than 100 access list entries. The next group of data in the test case is the simulated packet traffic data. Each of these lines will be at most 100 characters. The test case is terminated by a blank line or end-of-file.

The input file is **h.dat**.

OUTPUT:

For each test case print 3 numbers: the total count of access list entries compared; the number of simulated packets resulting in the 'permit' action; and the number of simulated packets resulting in the 'deny' action. Put no leading spaces before the first number and exactly 1 space before the 2nd and 3rd numbers.

See the sample input and output on the next page.

ACM Pacific NW Region Programming Contest

11 November 2000

Sample input:

```
* 204.69.0.0 255.255.248.0 * 0.0.0.0 0.0.0.0 * permit
* 0.0.0.0 0.0.0.0 * 0.0.0.0 0.0.0.0 * DENY

204.69.6.230 3072 24.115.85.245 23 6
192.168.215.17 53 204.248.52.7 1243 17
204.69.12.21 2118 152.163.241.11 21 6

* 0.0.0.0 0.0.0.0 * 0.0.0.0 0.0.0.0 * deny

192.102.199.19 27961 63.20.61.96 65535 17

6 198.138.176.100 255.255.255.255 * 204.69.4.100 255.255.255.255 111
permit
6 0.0.0.0 0.0.0.0 * 204.69.0.0 255.255.240.0 111 deny
6 0.0.0.0 0.0.0.0 * 204.69.0.0 255.255.248.0 80 permit
* 0.0.0.0 0.0.0.0 * 204.69.0.255 255.255.248.255 * deny
* 204.69.0.0 255.255.248.0 * 0.0.0.0 0.0.0.0 * deny
* 127.0.0.0 255.0.0.0 * 0.0.0.0 0.0.0.0 * deny
* 0.0.0.0 0.0.0.0 * 204.69.7.240 255.255.255.240 9990-9999 deny
* 0.0.0.0 0.0.0.0 * 0.0.0.0 0.0.0.0 * permit

204.69.4.87 80 204.69.9.12 6776 6
134.79.112.65 2186 204.69.4.218 21 6
216.112.217.140 2212 204.69.5.255 22 6
198.138.176.100 1053 204.69.4.100 111 6
198.138.176.100 1054 204.69.5.100 111 6
```

Sample output:

```
5 1 2
1 0 1
20 2 3
```

Hill Climbing for Poker Solitaire: A Case Study on the Impact of Neighborhood Size

Matthew Merzbacher

Mills College, Oakland CA 94613, USA
`matthew@mills.edu`

Abstract. This paper presents the results of an empirical study on the use of broadening and heuristics to improve search. We compare the performance of hill-climbing algorithms with different search breadth. Broadening the alternatives for hill-climbing yields small improvement in search results and is not generally worth the performance penalty incurred. In fact, evidence shows that broadening can degrade results. By contrast, even simple heuristics show great improvement, especially when combined with broadening, leading to conclusions about how to spend limited search resources most effectively.

Keywords: hill-climbing, heuristic search, broadening, gradient ascent

1 Introduction

Gradient ascent, also known as *hill-climbing*, is a fundamental search technique that operates efficiently without needing much memory. Starting with a random position, the best alternative move to a “neighboring” position is taken iteratively until no improvement is possible. With luck, the best possible position, or *optimum*, is found. The problems with gradient ascent are well known, including:

Plateau which has no gradient, so that all neighbors are equal.

Ridge where no single step improves things, but where a small step down can be followed by a larger step up.

Local Maximum where all nearby positions are worse, but which falls short of the optimum.

Steep Optimum where the best solution is surrounded by relatively unpromising territory, making it unlikely to be encountered.

Several solutions to these problems exist, including *random restart* and *simulated annealing* [2, 3]. Random restart repeatedly runs the hill-climbing algorithm from randomly generated starting positions, with the hope that at least one of the trials will start on the optimal hill. Simulated annealing allows some non-increasing steps to try to escape local maxima, ridges, and plateaus.

Neither random restart nor simulated annealing will usually be lucky enough to solve the steep optimum problem. The best solution in this case is to have some *heuristic* estimate to guide the search. The heuristic estimates the long-term promise of the particular position, rather than its actual score. Thus, the area around the steep optimum, which scores low, can still “look good” heuristically.

Each of these techniques (hill-climbing, simulated annealing, heuristic evaluation) can be augmented by *broadening* the search. Instead of considering just a few neighbors at each iteration, the neighborhood is extended outward to include more candidates. There are several variants of broadening, including macros [4], which consider selected promising more-distant neighbors based on past results. The ultimate broadening considers all possible solutions and will locate the optimum, but with unacceptable consequence to the search space size. Limited broadening may help avoid the problems associated with gradient ascent without overly slowing search.

In order to study the effect of broadening on hill-climbing algorithms, we use a problem with a large search space, many local maxima, and with a steep optimum.

2 The Problem

The particular problem, inspired by a magazine contest [1], uses the game “poker solitaire”. In this game, twenty-five cards are dealt from a standard deck in a five-by-five grid. Each row, column and the two diagonals of the grid are evaluated as poker hands, scoring a varying number of points. The goal is to reposition the cards¹, maximizing the sum of the twelve scores (five rows, five columns, two diagonals). The score for each hand is:

- pair (1 point) two of the five cards have the same rank.
- two pair (3 points) two pairs amongst the five cards.
- three of a kind (5 points) three of the five cards have the same rank.
- straight (7 points) the five cards may be arranged in sequence, e.g., 8-6-9-7-5. Aces may be high (above King) or low (below 2).
- flush (10 points) All five cards are the same suit.
- full house (12 points) one pair and three of a kind.
- four of a kind (25 points) four of the five cards have the same rank.
- straight-flush (50 points) both a straight and a flush.

The basic hill-climbing algorithm starts with a random grid of twenty-five cards and then swaps pairs of cards in the grid, as long as the score continues to improve. This algorithm can be broadened allowing consideration of more than two cards at a time for swapping. However, the number of neighbors when k cards are swapped is found by determining the possible combinations of k cards taken from twenty-five and multiplying that by the number of ways those cards can be ordered. For example, when swapping three cards (A, B, and C), A may replace B which replaces C, or A may replace C which replaces B. Removing symmetries, the number of possible moves involving k cards is:

$$B(k) = \frac{25!}{k(25-k)!}$$

¹ In the original game of poker solitaire, cards are placed one at a time and may not be moved. The whole point of our game is to allow movement of cards in the grid.

which grows formidably. $B(2) = 300$, $B(3) = 4600$, and $B(4) = 75900$. Thus, the benefit of broadening will need to be remarkable to outweigh its enormous cost.

This problem also often has a steep optimum. This can be seen by observing that four cards in a straight-flush combined with one bad card scores no points, while swapping that one mismatched card may immediately lead to fifty more points. Even a simple four-flush (four of five cards with the same suit) scores no points, while one swap will score ten. To help solve the steep optimum problem, we employ a simple heuristic estimate with positive values for promising hands that would otherwise score less. Choosing particular values is difficult, since the promise may not pay off, but given the immense relative value of a straight-flush, a four-straight-flush is quite promising and even a three-straight-flush is worth consideration. Experimentally, we demonstrate the value of this simple heuristic in conjunction with broadening.

3 Experiments

To measure the cost and value of broadening and heuristics, 1038 sets of twenty-five cards were randomly selected. For each set, one hundred random starting grids were generated. For each random grid, we ran both regular and heuristic hill-climbing with $k = 2$ and $k = 3$. In the $k = 3$ case, we selected the best choice from all three-way *or* a two-way swaps (thus, the branching factor is really $B(2) + B(3) = 4900$). In the case of heuristic hill-climbing, after the heuristic stops improving we took measurements and continued with the regular hill-climbing algorithm to termination. Thus, there are six variants:

1. hill-climbing with $k = 2$
2. hill-climbing with $k = 3$
3. heuristic hill-climbing with $k = 2$
4. heuristic hill-climbing with $k = 3$
5. heuristic hill-climbing, followed by hill-climbing, both with $k = 2$
6. heuristic hill-climbing, followed by hill-climbing, both with $k = 3$

We measured the distributions for both scores and depths along with statistics about how frequently each version of the algorithm yielded the optimum. Since there is no way to determine the true optimum, we used the best result found on any run by any variant. It is possible that obscure optimums could elude detection.

Fig. 1 compares the final score distributions for each of the six variants (average scores for each method are listed in Fig. 2). Qualitatively, the curves are in three distinct pairs – hill-climbing, heuristic hill-climbing, and heuristic hill-climbing with follow; broadening the search has negligible qualitative effect. The average score increase due to broadening is between four and five points, representing an improvement around five percent.

Further, we discovered that broadening to $k = 3$ only outperforms $k = 2$ in 57% of the hill-climbing runs, with similar results for heuristic variants. In the

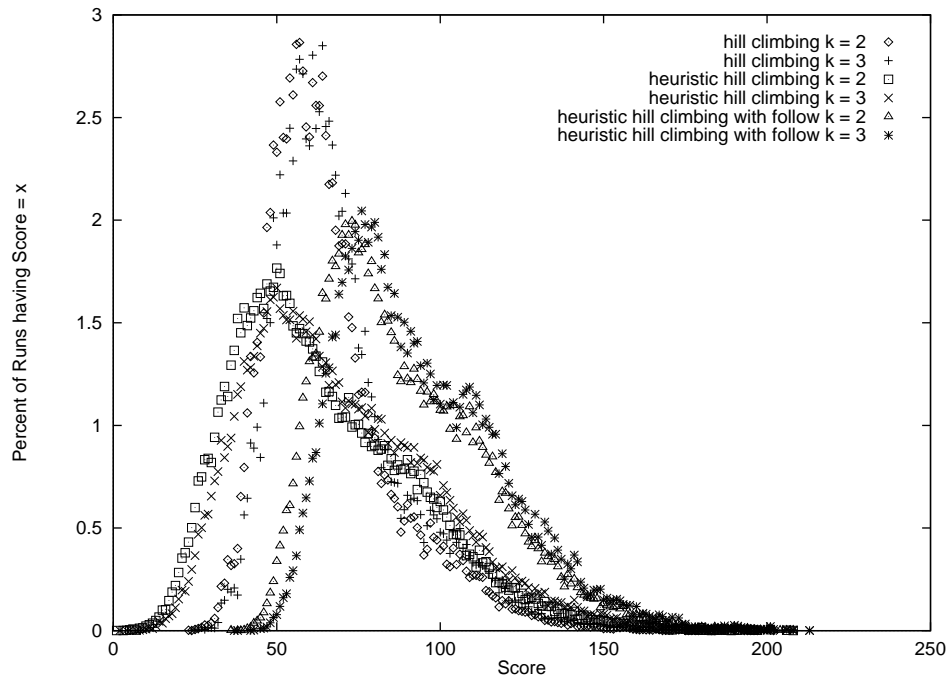


Fig. 1. Score distribution for six versions

other 43%, the narrower hill-climb found as good or better a result. Broadening may not help and was harmful 27% of the time!

Another surprise is that while the heuristic performs poorly on its own, with the followup of hill-climbing it is by far the best. Thus, a very simple heuristic can lead to a spot where blind search can follow up successfully.

method	avg. score	avg. depth	avg. cost	win %
hill-climbing $k = 2$	65.91	9.65	2895	1.15%
hill-climbing $k = 3$	69.36	7.54	36946	6.35%
heur. hill-climbing $k = 2$	64.61	12.61	3783	5.39%
heur. hill-climbing $k = 3$	69.65	10.02	49098	10.11%
heur. hill-climbing w/ follow, $k = 2$	89.01	16.89	5067	34.26%
heur. hill-climbing w/ follow, $k = 3$	93.84	13.39	65611	46.20%

Fig. 2. Algorithm Measurements

The point of the problem, however, was not to find the best average score, but to find the optimum from as many different starting deals as possible. For this task, broadening and heuristics both prove useful, as shown in the final column of Fig. 2. In all variants, broadening leads to more chance of locating the

optimum over a run of one hundred tests. Further, this chance rises substantially more when the heuristic was followed by basic hill-climbing.

Let us turn to measuring the depths of the search. The deeper a search goes, the more it costs. Fig. 2 shows the average depth and estimated average cost for each method. The cost is determined by multiplying the depth by the branching factor, since only the single best choice is evaluated at each step. The immense cost of broadening becomes clear here, as the cost with $k = 3$ is approximately thirteen times the cost with $k = 2$. Instead of paying the computational price of broadening, we could use that time for more random restarts. Over a thousand random runs with $k = 2$ can be run for the same price as one hundred runs with $k = 3$.

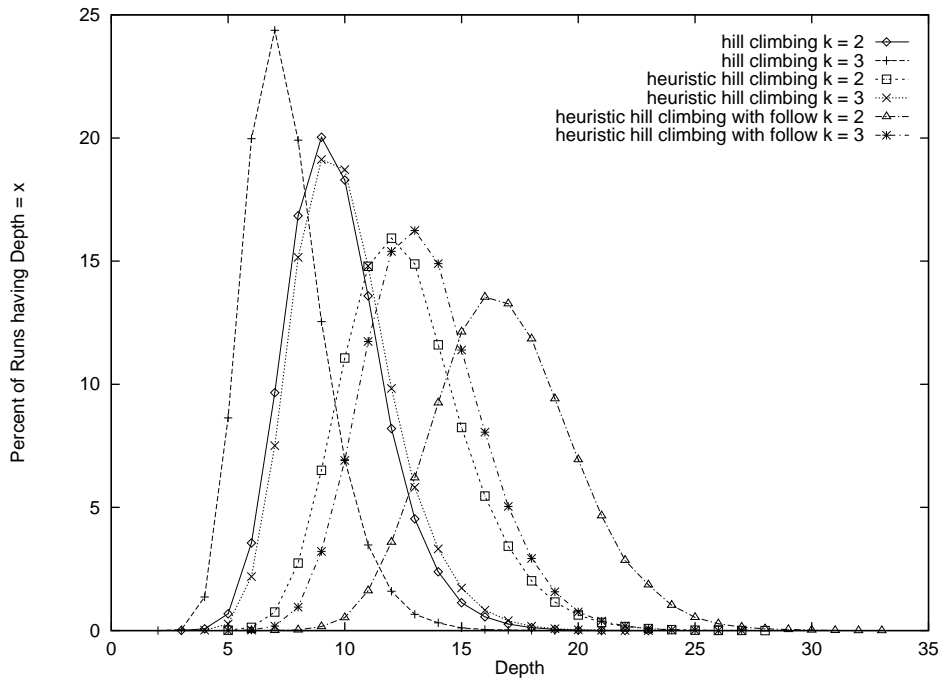


Fig. 3. Distribution of Number of Iterations

Fig. 3 shows the distributions of the number of iterations for each variant. In each scenario, broadening offers approximately twenty percent reduction in the number of iterations, not nearly enough to overcome the additional cost of evaluating so many alternatives.

4 Conclusion

The evidence suggests taking a pessimistic view of broadening. In general, a few well-chosen alternatives [6] will outperform a broad search of all close al-

ternatives. Broadening by a factor of twelve offered about a ten percent overall improvement in the quality of search results. Meanwhile, even a simple heuristic with $k = 2$ dominated searching with $k = 3$.

In practice, the solution to the particular original magazine problem was solved on a Sun SPARCstation 5 taking about a week. With broadening to $k = 3$, the optimal solution was found in a few days, and with heuristics the algorithm took less than an hour. There were several hundred successful entrants to the magazine contest (perhaps calculated by hand, perhaps by computer), so our entry did not win.

Since submission of this paper, we have also used poker solitaire as a homework problem in our introductory artificial intelligence class. The students were required to determine the cost of an optimal solution using breadth-first search and then to identify a promising heuristic. The student heuristics were considerably varied and under comparison in a follow-up experiment.

Improvement due to broadening and improvement due to the use of heuristics seems to be nearly independent. This suggests that broadening could be used in conjunction with heuristics for promising search runs, but not for all cases.

The impact of broadening must also enter into calculations when considering techniques such as macros, which exchange breadth for depth. Broadening can be worthwhile, but must be done with care and discrimination to be useful.

References

1. "Contest: Poker Solitaire II", *Games Magazine*, November 1999.
2. R. Greiner, "PALO: a probabilistic hill-climbing algorithm," *Artificial Intelligence*, vol. 84, no. 1-2, pp 177-208, 1996.
3. S. Kirkpatrick, C. Gelatt, and M. Vecchi, "Optimization by Simulated Annealing," *Science*, 220:671-680, 1983.
4. R. Korf, "Space-Efficient Search Algorithms," *ACM Computing Surveys*, 27(3):337-339, 1996.
5. J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Reading MA: Addison-Wesley, 1984.
6. M. Yokoo "Why adding more constraints makes a problem easier for hill-climbing algorithms: analyzing landscapes of CSPs," *Principles and Practice of Constraint Programming - CP '97*, pp. 356-70, Springer-Verlag, 1997.